Transforming Standard Process Models to Decentralized Autonomous Entities

Pieter Hens[1], Monique Snoeck[1], Manu De Backer[1,2,3,4], and Geert Poels[2]

[1] K.U.Leuven, Dept. of Decision Sciences and Information Management, Naamsestraat 69, 3000 Leuven, Belgium
[2] Universiteit Gent, Dept. of Management Information and Operations Management, Tweekerkenstraat 2, 9000 Gent, Belgium
[3] Universiteit Antwerpen, Dept. of Management Information Systems, Prinsstraat 13, 2000 Antwerpen, Belgium
[4] Hogeschool Gent, Dept. of Management and Informatics, Kortrijksesteenweg 14, 9000 Gent, Belgium

Abstract. Today, in state of the art process engine architectures, process models are executed by a central orchestrator (i.e. one per process). There are however a lot of drawbacks in using a central orchestrator, including a single point of failure and performance degradation. Decentralization algorithms that distribute the workload of the central orchestrator exist, but the result still suffers from a tight coupling between the different decentralized orchestrators and therefore has a decreased scalability. In this paper, we show practical transformations to decentralize a process model into autonomous, independent process engines. This solves the fundamental problems of the classical decentralization algorithms, increases the availability of the global process flow and makes it easier to re-specify and redeploy process models.

1 Introduction

In the last couple of years, process modeling received increasing attention from researchers and practitioners. Especially with the arrival of service oriented computing, process modeling became even more important. Starting from atomic services, new aggregate services can be built by combining the atomic services and describing an execution flow between the different entities. This way composite services are created, which can again be used in other compositions. When these compositions are described with a specific executable language (e.g. BPEL4WS [1] or BPMN 2.0 [2]), automated enactment using a process engine can be accomplished. The description of the process flow can be interpreted by a process engine (or orchestrator), which coordinates and triggers the described work.

Typically, the execution of each composite service (or process) is coordinated by one central entity (Fig. 1a, coordinator C0). This central orchestrator is initiated upon a request from a client and starts the execution of the workflow logic described in the composite service (Fig. 1a, tasks T1, T2 and T3). This is called CENTRALIZED ORCHESTRATION [3].

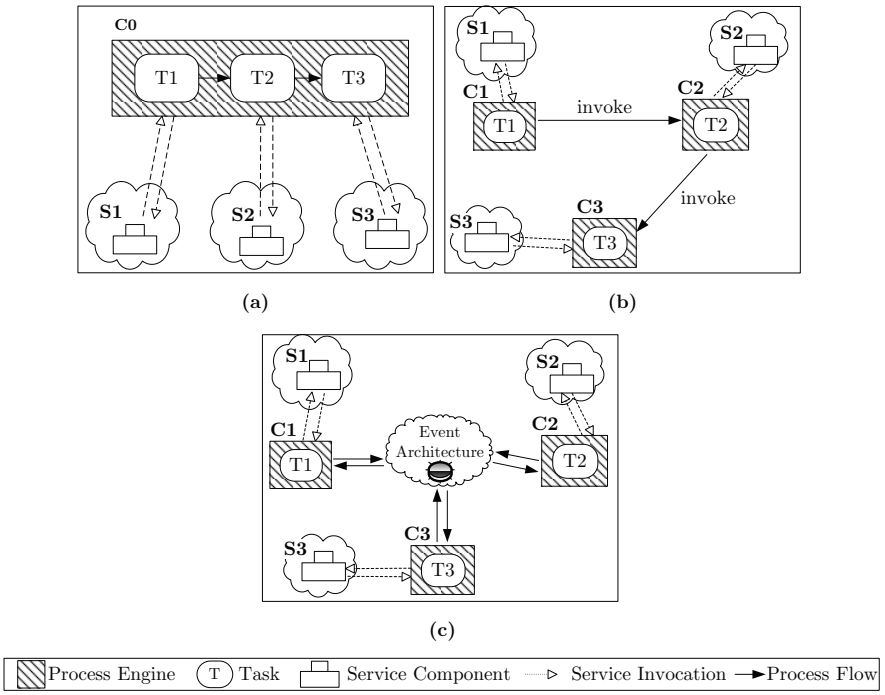


Fig. 1. Centralized, Decentralized and Event-Based Orchestration

The use of a central orchestrator per process struggles, however, with major problems in today's highly decentralized world. Using a central orchestrator (or execution engine) for a composite service creates: (a) *a single point of failure*, the services (work items) are distributed and decentralized (Fig. 1a: S1, S2 and S3), but the decision logic and coordination of the workflow is still located at one point (Fig. 1a: C0). Failure of the coordinator means failure of the entire process, even if the services are still available; (b) *unnecessary network traffic*, all (data) traffic from- and to- services invoked by the orchestrator runs through this central orchestrator, even if the data is of no importance to the orchestrator itself (e.g. data from S1 to S2 in Fig. 1a); and (c) *a performance bottleneck*, the number of process instances can run up very quickly and if all are coordinated at one point in the IT infrastructure, performance decreases significantly [4,5,6,7].

To overcome this bottleneck, solutions are given to decentralize the coordination work [4,5,6]. This results in separated process engines, which remove the need for a central orchestrator and decentralize the workflow logic (Fig. 1b, process engines C1, C2 and C3). This is DECENTRALIZED ORCHESTRATION.

Simple decentralization of the process flow fixes the fundamental problems of centralized orchestration (single point of failure and performance degradation), but not to a full extend [7]. Execution engines are still mutually tightly coupled in the process enactment infrastructure. For example, the start of execution engine C2 in Fig. 1b relies on its invocation by execution engine C1. C2 isn't autonomous and has to rely on decisions made by C1 (i.e. its request to start

C2). The logic of the next step in the process is located with the caller (C1), and not with the callee (C2). This tight coupling creates inflexible IT infrastructures and decreases scalability of the process architecture [8].

To solve the tight coupling, we proposed an extension to decentralized orchestration, which uses an event-based architecture as the communication paradigm in decentralized orchestration (see Fig. 1c) [7]. Decentralized Event-Based Orchestration will create autonomous process engines, capable of assessing their environment and deciding on their own when to initiate their execution (which is a useful property in process management [9]). It also creates a highly loose coupled infrastructure, which makes changes to the process flow relatively easy ('plug and play' of process engines). Notice that we've introduced an event driven architecture to support the decentralization of the process flow (full arrowhead arrows in Fig. 1), not for the invocation of services (open arrowhead arrows in Fig. 1), which has already been accomplished by many others (SOA and EDA [10]).

To gain the benefits from decentralized event-based orchestration, fundamental transformations of the modeled process flow are necessary. In this paper we'll introduce the practical transformations involved in transforming a standard (global) process to a decentralized event-based orchestrated process. The outcome of this transformation is a process model that can be executed by several event-based process engines (or orchestrators). Each orchestrator will be autonomous and distributed, which increases scalability and removes the single point of failure.

In the next section we briefly explain decentralized event-based orchestration, followed by the positioning of the transformations in process development and enactment (Sect. 3). In Sect. 4 we show the actual transformations involved in transforming a process model to a decentralized event-based model. In Sect. 5 we end the paper with a conclusion and some implications of this research.

2 Decentralized Event-Based Orchestration

Decentralized event-based orchestration is the coordination of a single process flow by multiple, autonomous orchestrators [7]. Each orchestrator coordinates a little piece of the global, entire process flow. Thus, combined, they coordinate the global process as modeled by the process modeler. Communication between the orchestrators is accomplished by means of an event based architecture. An event based architecture is a communication paradigm that uses a publish/subscribe interaction scheme. An event is something that happens, and when an event occurs, a notification of this event occurrence is published in the architecture. The architecture then routes this notification to interested parties (the subscribers).

Using a publish/subscribe interaction scheme accomplishes loose coupling between two communicating entities. These include: space decoupling (unawareness of interaction partners), time decoupling (interaction partners don't need to be active at the same time) and synchronization decoupling (asynchronous send and receive) [11,7]. Using an event based architecture for decentralized orchestration

thus removes the tight coupling between the different distributed execution engines, which makes the process architecture more scalable. New process engines which consume already published events, can simply be added to the process architecture without making any changes to the already running infrastructure. Note that the supporting entities in an event based architecture (the *cloud* in Fig. 1c) are also loosely coupled and don't add another single point of failure. Many solutions exist that distribute the event based architecture itself [8].

A second consequence of using an event driven architecture in a process decentralization setting is that each execution engine becomes autonomous. A decentralized orchestrator can asses its environment, and when the environment is in a certain state (i.e. some specific events happened), it starts its execution. An orchestrator doesn't rely anymore on messages that request its initiation, in stead it decides for itself when to start. After execution, the orchestrator publishes a notification of its occurrence. This event alters the environment, whereupon other orchestrators *may* react and execute their process logic. A chain of these event publications and consummations (assessment of the environment) results in the execution of the process flow as modeled by the process modeler.

3 Deploying a Process Specification

Transforming a process model to a decentralized event-based model happens at deployment time. This enables the process modeler to not take decentralization into account when designing the process model. Figure 2 shows a process specification-deployment structure. First a process modeler designs a global, fully specified, executable process model. This model not only specifies the flow, but also specifies which service(s) will handle which tasks defined in the process flow (service invocations, see top part of Fig. 3). After process specification, the process can be deployed. It is at this time that the specified process will undergo a transformation which decomposes the process into smaller parts. Note again that our decentralization focuses on transforming the process logic, not the invocation of services.

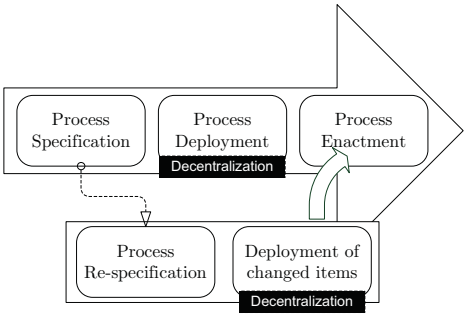


Fig. 2. Deployment and re-specification of a process model in an event-based orchestration

To decentralize the process flow, a unit of decomposition has to be chosen. The unit of decomposition can be anything from a *task* to a group of process elements (*tasks*, *gateways*, ...). Each unit of decomposition will be deployed to a separate execution engine, resulting in a one-to-one mapping between unit of decomposition and coordinator. This is illustrated in Fig. 3, with the unit of decomposition a *task*. Each task in the original process flow (T1, T2, T3 and T4) is to

be deployed on its own process engine (C1, C2, C3 and C4).

After deployment, re-specification and redeployment of the process model can be done with little effort. The process modeler can re-specify the global process model, after which only the changed items in the process flow will need to be redeployed (see Fig. 2). The already existing not-changed items can be left running without interruption. This is possible due to the space decoupling property of event-based orchestration [7].

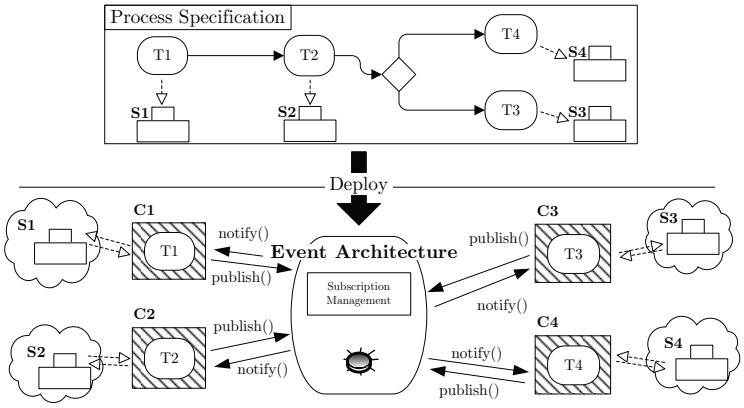


Fig. 3. Deploying a process specification to a decentralized event-based orchestration

4 Transformation

To illustrate the transformations involved in deploying a process specification to a decentralized event-based orchestration, we use BPMN 2.0 [2] as the notation in which the process is specified. Besides a workflow notation, BPMN 2.0 has a token based execution semantics. This makes it possible to directly execute process models defined in BPMN 2.0. Process engine solutions like jBPM [12] and Activiti [13] already support this feature. Because BPMN 2.0 defines a vast amount of entities that can occur in the process model, we define the scope for our transformations to the Standard Process Models as defined by [14]. A Standard Process Model embodies *process elements* which are connected to each other by *transitions*. A process element is either an activity, an AND-Split, a XOR-Split, an OR-Split, an AND-Join and a XOR-Join. These elements correspond to the basic control flow patterns, together with the multi-choice control flow pattern [15].

The outcome of our decentralization is also compliant with the BPMN 2.0 metamodel. This way, any BPMN 2.0 process engine can eventually run the decentralized process model (if it supports the used concepts and a publish/subscribe communication architecture).

4.1 Translating the Unit of Decomposition

We choose a *task* as the unit of decomposition in our transformation. This guarantees a fine grained decentralization. Each task gets translated to a separate process containing that same original task, (multiple) start event(s) and one end event (see Fig. 4 for an example). The unique end event indicates the notification of the tasks accomplishment. This end event is transcribed in BPMN 2.0 as a *throw signal event*. Signal events indicate events that are not process bound, multiple processes can have start events that are triggered from the same broadcasted signal. The semantics of a signal-event resemble closely an event-notification of an event-based architecture. They are thus the most appropriate notation to symbolize our loosely coupled event structure.

As start for the new process, event rules need to be calculated. An event rule is a rule stating in which situation this new process can start. For example, the start rule `(EventA AND EventB) XOR (EventC)`, simply says that the process starts after the occurrence of event A and event B or after the occurrence of event C. For each task, this rule is deduced from the original global process model (the input for the transformation). Event rules are transcribed by using a *catch signal event*. A conjunction in the event rule is indicated by displaying the start event with a *parallel multiple* marker. Disjunctions in the event rule are denoted by using multiple start events. This notation ensures that an event rule is always expressed in a Disjunctive Normal Form (DNF). Figure 4 shows an example of the resulting new process for one unit of decomposition (task).

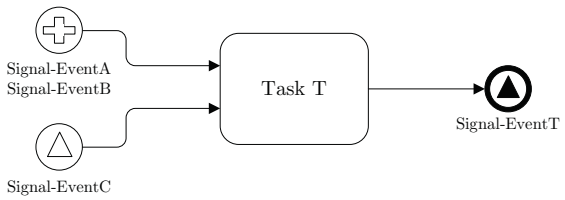


Fig. 4. Result of a single task after the decomposition of a process flow

4.2 Event Rules

To find the start event rule of a task, the preceding elements in the process flow have to be investigated:

Sequence Flow. The most basic start rule for a task is that it can only start after the successful completion of the preceding task in the sequence flow. Figure 5a shows the corresponding transformation. Task X starts after the completion of task A. Our unit of decomposition is a task, thus it is put in a separate process, with as start event a catch of the signal indicating the end of task A.

Exclusive Gateway. If the incoming flow of a task is connected with an exclusive gateway, the start of the task is dependent on the successful execution of one of the tasks preceding the exclusive gateway. The event rule for a task connected with an exclusive gateway is thus a disjunction of the signals indicating the completion of the process elements preceding the exclusive gateway. Figure 5b shows this transformation.

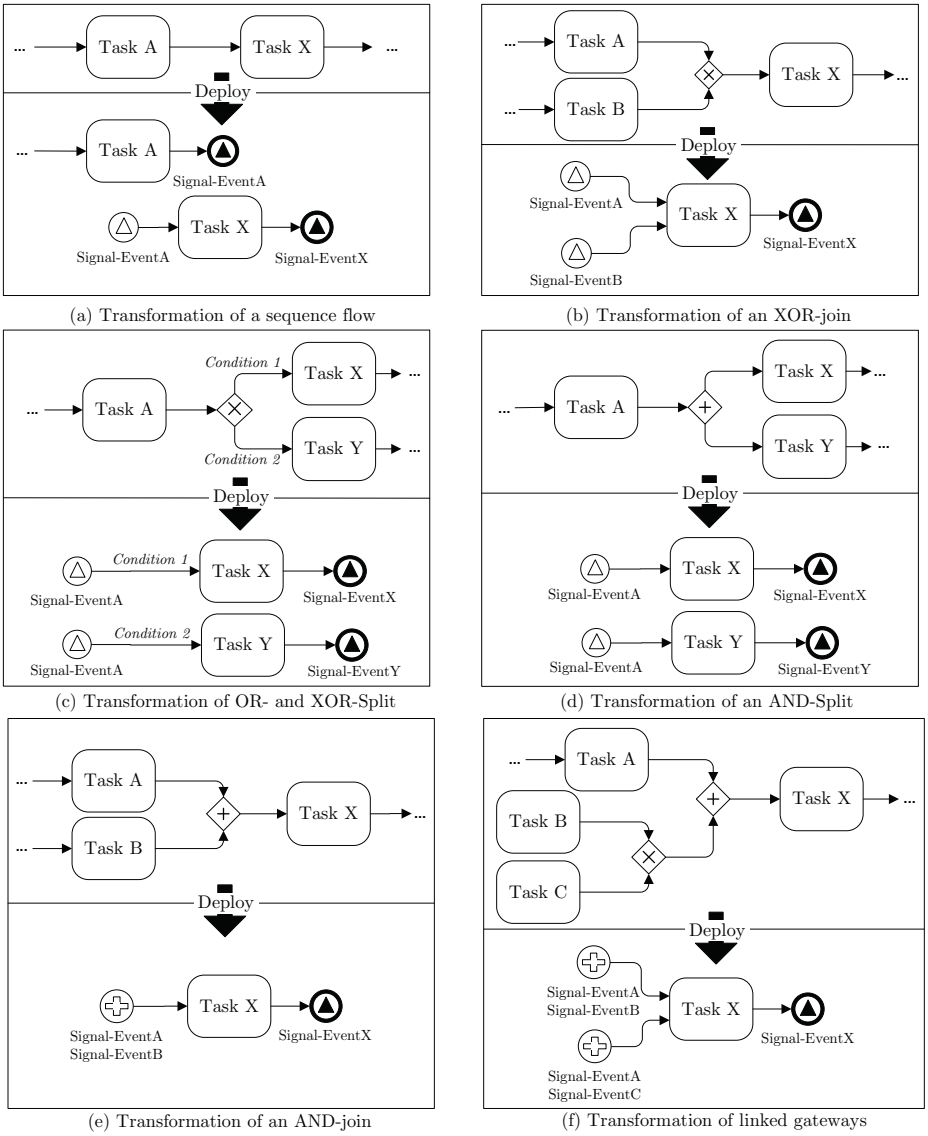
Conditions on preceding sequence flows. BPMN 2.0 states that the conditions belonging to an OR- and XOR-split are put on the sequence flows succeeding the OR- and XOR- gateway (*conditional flow*). If the condition is valid, that specific path in the process is 'chosen'. These conditions should be conveyed to the newly created decentralized process and displayed on the correct sequence flow (see Fig. 5c). This means that the (decentralized) task will only start when the environment is in a certain state (i.e. some events happened) and when the condition on the respective sequence flow is true. It is possible that, when multiple exclusive gateways are linked together, multiple conditions should be true before the task can start. All these conditions are put in conjunction on the respective sequence flow.

Parallel Gateway. Figure 5d and 5e illustrate the transformation for a task with its incoming sequence flow connected to a parallel gateway (either AND-split or AND-join). The event rule becomes a conjunction of the completion-notifications of the process elements preceding the parallel gateway. Observe that for an AND-split only *one* signal event (`Signal-EventA`) is used to trigger the multiple catch events (for tasks X and Y). This reduces the number of different event messages that need to be exchanged in the decentralized orchestration, compared to creating a separate signal event for each flow outgoing the AND-gateway.

Linked Gateways. Gateways can also be directly linked together by sequence flows (see Fig. 5d). In this situation the event rule has to be calculated recursively according to the transformations described above. The eventual rule is put in DNF so that it can be represented in the BPMN schema (see Sect. 4.1). Figure 5d shows an example. Following the transformations stated above results in an event rule for task X: $TaskA \wedge (TaskB \vee TaskC)$ and in DNF: $(TaskA \wedge TaskB) \vee (TaskA \wedge TaskC)$.

4.3 Formal Implementation

We have implemented the transformation in the Atlas Transformation Language (ATL) [16]. ATL is available as a plugin in the Eclipse Modeling Framework and provides a way to declaratively describe the transformation of a source model (supported by a metamodel) to a target model. Figure 6a shows the transformation structure. As input, the transformation takes a source model which conforms to the BPMN 2.0 metamodel (any BPMN Diagram Interchange file [2]). The output of the transformation is also a model conforming to the BPMN 2.0 metamodel. The output file (a BPMN Diagram Interchange file) can be directly uploaded in any process engine supporting BPMN 2.0. If the



(a) Transformation of a sequence flow

(b) Transformation of an XOR-join

(c) Transformation of OR- and XOR-Split

(d) Transformation of an AND-Split

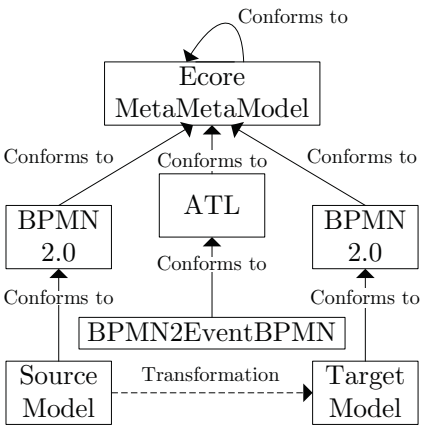
(e) Transformation of an AND-join

(f) Transformation of linked gateways

Fig. 5. Deploying Standard Process Flow elements to a decentralized event-based orchestration

engine implements signal event communication in a publish/subscribe fashion, the benefits described in [7] will become available.

A small excerpt of the ATL transformation code is found in Fig. 6b. The transformation contains only one matching rule[1] which translates a task in the source model to a new resulting process as described in Sect. 4.1.



(a) ATL transformation structure

```
rule Task2Process {
 from
  s: BPMN20!Task
 to
  newTask: BPMN20!Task (
   id <- s.id,
   name <- s.name
  ),
  endev: BPMN20!EndEvent (
   id <- 'EndEvent' + s.id,
   eventDefinitions <- outputDef
  ),
  outputDef: BPMN20!SignalEventDefinition (
   id <- 'outputDef' + s.id,
   signalRef <- outputSignal
  ),
  outputSignal: BPMN20!Signal (
   id <- 'Signal' + s.id),
  ),
  process: BPMN20!Process (
   id <- 'sid-Process' + s.id,
   flowElements <- Sequence{newTask},
   flowElements <- Sequence{outputSeqFlow},
   flowElements <- Sequence{endev},
   flowElements <- s.getEventRule()
    -> collect(e | thisModule.createStartEvents(e))
  )
```

(b) Excerpt of the ATL BPMN2EventBPMN transformation code

Fig. 6. ATL Transformation

5 Conclusion, Implications and Future Research

In this paper we introduced the practical transformations necessary to transform a standard process model to a decentralized event-based orchestration.

Using an event based communication paradigm in a decentralized orchestration creates highly flexible, autonomous entities, which increase scalability and availability of the process flow. By doing the transformations at deployment, the process modeler doesn't need to know the decentralization details. Deployment of a changed process flow can also be done fairly quickly, without the need to interrupt the current (unchanged) process entities.

Another implication of working with an event based architecture in process enactment is that event-logs of the running processes become readily available.

[1] A matching rule is a rule that matches an entity from the source model to new entities in the target model

This enables easier access to process mining [17] or complex event processing [18]. Yet another application can be agent based development, where the operations of the agents are noted in a process flow-like style.

Future research includes widening the scope of the transformable process elements to a level 2 process modeling [19], which includes subprocesses, intermediate events, transactions, ... as well as including data management. We also intend to prove the correctness of the transformation rules with process algebra and formally validate the added value by testing on availability (stress testing) and scalability of the decentralized event-based process flow.

References

1. Oasis: Web service business process execution language version 2.0. Oasis Standard
2. Object Management Group: Draft proposal for: Bpmn 2.0, beta 2. http://www.omg.org/cgi-bin/doc?dtc/10-06-04 (May 2010)
3. Barros, A., Dumas, M., Oaks, P.: Standards for web service choreography and orchestration: Status and perspectives. BPM Workshops 61–74
4. Chafle, G., Chandra, S., Mann, V., Nanda, M.: Decentralized orchestration of composite web services. Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (2004) 134–143
5. Muth, P., Wodtke, D., Weissenfels, J., Dittrich, A., Weikum, G.: From centralized workflow specification to distributed workflow execution. Journal of Intelligent Information Systems **10**(2) (1998) 159–184
6. Nanda, M., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. ACM SIGPLAN Notices **39**(10) (2004) 170–187
7. Hens, P., Snoeck, M., De Backer, M., Poels, G.: Decentralized Event-Based Orchestration. In: Third International Workshop on Event-Driven Business Process Management. (2010)
8. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer-Verlag New York, Inc. Secaucus, NJ, USA (2006)
9. Jennings, N., Norman, T., Faratin, P., OBrien, P., Odgers, B.: Autonomous agents for business process management. Applied Artificial Intelligence **14**(2) (2000)
10. Michelson, B.: Event-driven architecture overview. OMG report (2006)
11. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Computing Surveys (CSUR) **35**(2) (2003) 131
12. jBPM. http://jboss.org/jbpm
13. Activiti. http://www.activiti.org/
14. Kiepuszewski, B., ter Hofstede, A., van der Aalst, W.: Fundamentals of control flow in workflows. Acta Informatica **39**(3) (2003) 143–209
15. van Der Aalst, W., Ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and parallel databases **14**(1) (2003) 5–51
16. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: the 21st ACM SIGPLAN symposium. (2006)
17. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow mining: A survey of issues and approaches. Data & Knowledge Engineering **47**(2) (2003) 237–267
18. Luckman, D.: The power of events: an introduction to Complex Event Processing. Addison-Wesley (2002)
19. Silver, B.: BPMN Method and Style. Cody-Cassidy Press (2009)